Advanced Parallel Programming

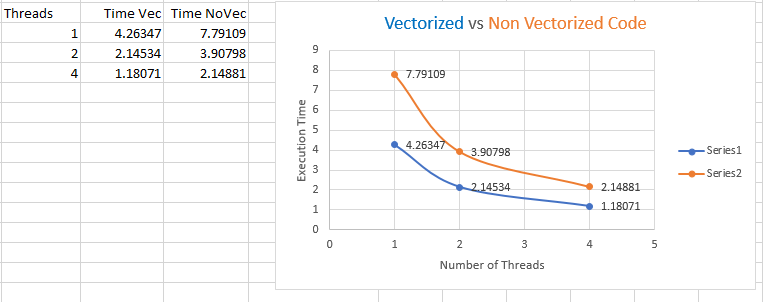
OpenMP : Vectorization and Hybrid Programming

Michel Herrera Sanchez

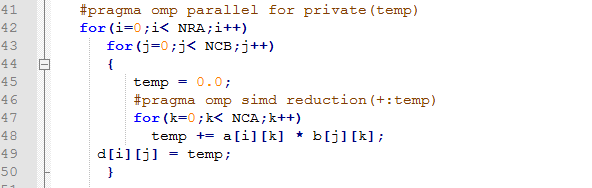
LABS1

Vectorization with OpenMP

1. As expected openmp vectorized code achieved an acceleration of 2x over pure openmp implementation.



Following the code (multf\_omp.c) implementation details:



External loop was parallelized considering temp variable as private for all omp threads (line 41). The most internal loop (line 46) was vectorized using the omp simd directive and applying a reduction on temp variable.

1. As shown bellow vectorized code is faster than non vectorized. Vectorized functions (saxpy, saxpyi) called on vectorized loops shows a better execution time of 1.116s over the non vectorized code with a minimum of 1.254s.

|  |  |
| --- | --- |
| Function | Execution Time (s) |
| saxpy\_no\_simd | 1.260 |
| saxpyi\_no\_simd | 1.254 |
| saxpy | 1.116 |
| saxpyi | 1.116 |

See code saxpy\_omp.c for implementation details.

1. Parallelizing the for loops of N iterations decreased the execution time 3 times fold. There is no doubt that the combination of parallelization and vectorization was the winner in respect to code optimization.

|  |  |
| --- | --- |
| Function | Execution Time (s) |
| saxpy\_no\_simd | 0.427 |
| saxpyi\_no\_simd | 0.432 |
| saxpy | 0.395 |
| saxpyi | 0.395 |

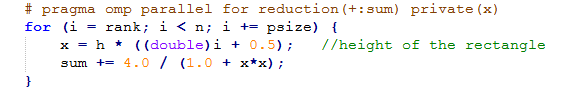
See code saxpy\_omp\_par.c for details.

LABS 2. Hybrid Programming

1. Parallelization of pi\_integral.c using Hybrid MPI/OpenMP

The parallelization of code pi\_integral.c mainly consists on split the intervals for the integral calculation equally among a set of process. The number of iterations is initially sent to each process using the function MPI\_Bcast from process with rank zero. Then the calculation of the integral is done on each process after the number of iterations is known. Afterwards the results of all partial calculations per process need to be globally reduced using a SUM operator and stored in process with rank 0. The global reduce operation was implemented calling the MPI function MPI\_Reduce. At the end of the program the process with rank 0 shows the results.

The OpenMP implementation is embedded in the calculation of the integral. See the following code to understand the parallelization with OpenMP of the integral calculation.



The pure MPI implementation provides the best execution time and speed up for 1000,000 iterations of pi\_integral.

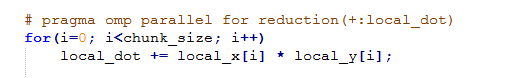
|  |  |  |
| --- | --- | --- |
| Topology | Execution Time (s) | Speed Up |
| Sequential | 0.013 |  |
| Pure MPI (16 process) | 0.00449 |  |
| Pure OpenMP (1, 16) | 0.005131 |  |
| Hybrid (2,8) | 0.004491 |  |
| Hybrid (4,4) | 0.004769 |  |
| Hybrid (8,2) | 0.003280 |  |
| Hybrid (16, 1) | 0.005530 |  |

For implementation details check the Hybrid MPI/OpenMP implementation in code pi\_integral\_mpiomp.c

1. Parallelization of dotprod.c using Hybrid MPI/OpenMP

The current solution assumes that the number of elements of each dotprod vector is divisible by the number of mpi process. The number of vector’s elements is initially sent to each process using the function MPI\_Bcast from process with rank zero. Data is initially loaded in process with rank zero for the two operands vectors x and y. Then, two MPI\_Scatter calls are performed using root process the one with rank 0. The first call to MPI\_Scatter distributes equals chunk sizes of portions of vector X among all process. The same happen with the second MPI\_Scatter call that distributes equally chunk sizes of vector Y among all the process. Right after receiving the X and Y chunks each process performs the dotprod multiplication with the portion of the vectors placed in his local memory. The process with rank 0 leads a global reduction operation calling method MPI\_Reduce to collect and sum all the partial dotprod results present on each of the process. Once it’s done the process with rank 0 print out the results.

See the following code to understand the parallelization of the partial dotprod with OpenMP.



The pure OpenMP implementation provides the best execution time and speed up for a dotprod of 16384 elements. This could be related with the exchange of data between the process.

|  |  |  |
| --- | --- | --- |
| Topology | Execution Time (s) | Speed Up |
| Sequential | 0.007 |  |
| Pure MPI (16 process) | 0.002491 |  |
| Pure OpenMP (1, 16) | 0.000400 |  |
| Hybrid (2,8) | 0.000562 |  |
| Hybrid (4,4) | 0.000677 |  |
| Hybrid (8,2) | 0.000861 |  |
| Hybrid (16, 1) | 0.024117 |  |

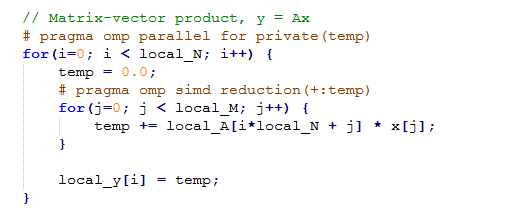
For implementation details check the Hybrid MPI/OpenMP implementation in code dotprod\_mpi\_omp.c

1. Parallelization of mxvnm with Hybrid MPI/OpenMP

The current solution assumes that the number of rows/columns of the matrix and equals to the number of elements of the vector. The number of columns of the matrix should be divisible by the number of mpi process.

Before partial matrix/vector multiplication happens on each of the participant MPI process. The number of rows and columns of the matrix are broadcast to all process using the MPI function MPI\_Bcast. Then each process receives a submatrix from the original matrix with the same number of columns but with a number of rows equals to number of rows of the matrix divided by the number of process. This was relatively easy to implement using the MPI\_Scatter function. A problem arises to send the same copy of vector X to all the process due to broadcast was not a suitable method. To send exactly the same copy of vector X to all process, we took the opposite approach. Each process initializes an equal size portion of the vector and then using the call to MPI\_Allgather it was possible to copy to each process exactly the whole vector X with all chunks coming from the different process. What follows is the partial multiplication of each submatrix privately stored on each process with the whole vector X. At the end process with rank 0 got the partial results from each process to fill the rows for the vector Y and final result. Each process produce one element of vector Y and it’s finally collected in process with rank 0.

The OpenMP implementation is embedded in the partial multiplication of the submatrix and the vector on each mpi process. See the following code to understand the parallelization with OpenMP.



The Hybrid MPI/OpenMP implementation provides the best execution time and speed up for the multiplication of an 8192x8192 matrix element elements with a vector of 8192 elements.

|  |  |  |
| --- | --- | --- |
| Topology | Execution Time (s) | Speed Up |
| Sequential | 0.256 |  |
| Pure MPI (16 process) | 0.100430 |  |
| Pure OpenMP (1, 16) | 0.509448 |  |
| Hybrid (2,8) | 0.350717 |  |
| Hybrid (4,4) | 0.176397 |  |
| Hybrid (8,2) | 0.339455 |  |
| Hybrid (16, 1) | 0.096907 |  |

For implementation details check the Hybrid MPI/OpenMP in code mxvnm\_mpi\_omp.c